Projet d'Architecture Interne des Systèmes d'Exploitation
Master 1 - Calcul Haute Performance et Simulation

Alan Vaquet et Killian Babilotte

15 Mars 2020

Sommaire

1 Introduction **2**

2 Support des fonctions d'allocation et préchargement de l'allocateur **2**

3 Méthode de compilation et système de build **3**

4 Mapping mémoire, alignement et métadonnées **4**

5 Recyclage de bloc et politique de sélection **6**

6 Autres optimisations et mutlithreading **8**

7 Tests de performances et profilage mémoire **9**

8 Conclusion **11**

References **11**

1 Introduction

Pour ce projet, nous avons du créé un allocateur mémoire fondé sur l'utilisation de deux fonctions principales: mmap et munmap. Il fallait également faire en sorte que l'utilisation et l'installation sous forme de librairie de notre allocateur soit aisée.
Les deux premières parties de ce compte rendu présentent la création de la librairie, l'interception des fonctions de la libc ainsi que la compilation et l'installation de cette librairie.
Ensuite nous présentons les bases du fonctionnement de notre allocateur avec l'utilisation de mmap et munmap, la technique permettant d'aligner les blocs et la structure des métadonnées individuelles à chaque bloc de mémoire géré par notre allocateur.
Une fois le fonctionnement basique décrit, c'est la politique de sélection et de recyclage qui sont présentés. Cette dernière permet de minimiser la consommation mémoire de notre allocateur tout en garantissant des durées d'allocation et de libération mémoire minimes.
Encore dans un objectif de performances et de solidité du code, une nouvelle partie de ce compte rendu abordera des optimisations apportés aux fonctions calloc, et realloc. Ainsi qu'au sécurités apportés pour sécuriser l'utilisation en multithreading.
Pour finir nous analyserons et comparerons les performances de notre allocateur et celles de la libc, puis présenterons les outils de profilages mémoire présent dans notre code.

2 Support des fonctions d'allocation et préchargement de l'allocateur

Nous avons adapté notre code de sorte à créer une librairie dynamique qui intercepte les appels aux fonctions d'allocation classique (malloc, calloc, realloc et free) au sein d'un code source lorsque celui inclus le header de notre librairie "mAlloK.h". Mais aussi pour qu'il soit possible de supplanter ces appels dans un exécutable qui charge les charge dynamiquement en préchargeant notre librairie via la variable d'environnement LD_PRELOAD.

Pour ce faire notre code repose sur un fichier source principal "mAlloK.c" qui contient toute les définitions de nos fonctions pour mettre en place notre allocateur. Ensuite pour intercepter les fonctions d'allocations seulement au seins d'un code source nous avons écrit notre header en utilisant l'astuce évoqué dans le chapitre 9 de [1] de tel sorte que le code de notre header est:

```
#ifndef _mAlloK_h
#define _mAlloK_h

#include <stdlib.h>
#include <malloc.h>

extern void *mAlloK(size_t);
extern void freeAK(void *);
extern void *cAlloK(size_t, size_t);
extern void *reAlloK(void *, size_t);

#define malloc(size) (*mAlloK)(size)
#define free(ptr) (*freeAK)(ptr)
#define calloc(nmemb, size) (*cAlloK)(nmemb, size)
#define realloc(ptr, size) (*reAlloK)(ptr, size)

#endif
```

Il déclare nos quatre fonctions qui sont elles définies dans notre fichier "mAlloK.c" ensuite linkées à la compilation par le linker à la place des symboles malloc, free, calloc et realloc. Le mot clé extern est essentiel ici pour que la définition des fonctions ne ce fasse bien qu'une fois dans notre fichier "mAlloK.c". L'avantage de cette méthode est de pouvoir faire co-exister plusieurs implémentations d'allocation sous le même symbole

malloc dans un même code.

Mais pour pouvoir préloader notre implémentations sur un code déjà compilé, nous avons dû créé un deuxième fichier source "mAlloK_preload_wrapper.c" :

```c
#include "./headers/preload.h"
#include <sys/types.h>

void *malloc(size_t size)
{
  return mAlloK(size);
}

void *calloc(size_t nmemb, size_t size)
{
  return cAlloK(nmemb, size);
}

void free(void *ptr)
{
  return freeAK(ptr);
}

void *realloc(void *ptr, size_t size)
{
  return reAlloK(ptr, size);
}
```

qui définie les symboles de malloc pour exécuter notre code. Le header "preload.h" déclare seulement nos symboles mAlloK pour ce fichier.

3 Méthode de compilation et système de build

Pour créer et compiler notre projet nous utilisons le système GNU Make avec le fichier Makefile suivant à la racine du projet :

```
DEFAULT_GOAL := default
CC?=gcc

default: setup lib preload

setup:
   mkdir -p ./build
   mkdir -p ./test/perf_test/dat

lib: setup ./src/mAlloK.c
   $(CC) -O3 -fPIC -shared -o ./build/libmAlloK.so ./src/mAlloK.c -lm -lpthread

preload: setup ./src/headers/preload.h
   $(CC) -O3 -fPIC -shared -o ./build/libmAlloK_preload_wrapper.so \
   ./src/mAlloK_preload_wrapper.c ./src/mAlloK.c -lm -lpthread

install: uninstall ./build/libmAlloK.so ./src/headers/mAlloK.h
   cp ./build/libmAlloK.so /usr/local/lib/
   cp ./src/headers/mAlloK.h /usr/local/include/
```

```
uninstall:
   rm -f /usr/local/lib/libmAlloK.so
   rm -f /usr/local/include/mAlloK.h
```

Il se charge de créer les deux fichiers de librairies dynamiques associé à notre projet, celui servant à l'appel par include qui peut être installé via la commande "make install" avec les droits root dans l'arborescence de fichier, et celui servant au préload.

4 Mapping mémoire, alignement et métadonnées

Dans notre allocateur, un chunk représente un sous espace mémoire réservé au programme. Chaque chunk correspond à un espace mémoire renfermant les métadonnées le concernant immédiatement suivi de l'espace mémoire qu'il représente, de cette facon chaque chunk a ses métadonnnées localement afin d'éviter une gestion trop globale de la mémoire.

Métadonées

```
// structure d'un chunk
typedef struct chunk_t chunk_t;

struct chunk_t
{
  size_t size_status;          // taille et status du chunk

  chunk_t *previous;           // pointeur vers le chunk precedent
  chunk_t *next;               // pointeur vers le chunk suivant

  chunk_t *previous_free;      // pointeur vers le chunk libre precedent
  chunk_t *next_free;          // pointeur vers le chunk libre suivant
};
```

Lorsque de l'espace mémoire d'une certaine taille est demandé à l'aide de malloc, calloc ou realloc et qu'un chunk libre de taille suffisante n'est pas disponible alors un chunk d'une taille minimum prédéfini est créé à l'aide de mmap.

Allocation de chunk

```
...
#define SIZE_MIN_CHUNK                  (256 * 1024)
...
#define mmap_chunk(size)        (mmap(NULL,                         \
                                      CHUNK_SIZE+size,              \
                                      PROT_READ|PROT_WRITE,         \
                                      MAP_PRIVATE|MAP_ANONYMOUS,    \
                                      -1, 0))                       \
...
chunk_t *chunk = (SIZE_MIN_CHUNK > total_size) ? mmap_chunk(SIZE_MIN_CHUNK) : mmap_chunk(total_size);
...
```

Si la taille demandée est supérieure à cette limite alors un chunk de cette taille est alloué et retourné. Sinon un chunk de cette taille limite est créé, puis découpé en un chunk alloué de la taille demandé et un chunk libre de taille restante. Le chunk alloué est retourné et le chunk libre sera redécoupé à son tour et ainsi de suite.

Découpage de chunk

```
...
static chunk_t *cut_chunk(chunk_t *chunk, const size_t size)
{
  chunk_t *new_chunk = (void*)chunk + size + CHUNK_SIZE;

  if(chunk->next != NULL){
    chunk->next->previous = new_chunk;}

  set_chunk(new_chunk, (get_size(chunk) - CHUNK_SIZE - size) | get_dirty(chunk), chunk, chunk->next, NUI

  set_chunk(chunk, size | STATUS_MASK | get_dirty(chunk), chunk->previous, new_chunk, NULL, NULL);

  return new_chunk;
}
...
```

Lorsque l'utilisateur appel free sur un espace mémoire donné par notre allocateur, soit le chunk qui lui correspond n'a pas de successeur ou de prédécesseur dans quel cas il représente un espace mémoire entier obtenu par un mmap et on utilise munmap pour rendre la mémoire au système. Sinon le chunk correspondant est libéré et ajouté aux chunks libres.

Libération de chunk

```
...
#define munmap_chunk(chunk)             (munmap(chunk, get_size(chunk) + CHUNK_SIZE))
...
if(only_chunk(chunk)){
    munmap_chunk(chunk);
}else{
    add_free_chunk(chunk);
}
...
```

Pour garantir l'alignement mémoire, chaque espace mémoire de taille t demandé par l'utilisateur est arrondi au multiple de la taille d'un mot système supérieur. Si la taille demandée est de 15 octets, et que les mots font 8 octets alors un chunk correspondant à un espace mémoire de 16 octets sera créé et cet espace mémoire lui sera retourné. Puisque l'adresse retournée par mmap est aligné sur un page système et donc sur un mot, et que la structure représentant les métadonnées de chaque chunk est elle aussi un multiple de mots systèmes, toutes le adresses manipulées et retournées par notre allocateur sont alignées.

Alignement mémoire

```
...
#define WORD_SIZE                       __WORDSIZE/8
...
#define roundTo(value,roundto)          ((value + (roundto - 1)) & ~(roundto - 1))
...
size = roundTo(size,WORD_SIZE);
...
```

5 Recyclage de bloc et politique de sélection

Comme expliqué précédemment lorsque l'utilisateur appel free sur un espace mémoire donné par notre allocateur et que ce dernier n'est qu'une potion d'un espace mémoire plus grand obtenu par découpages successifs du chunk d'origine, alors il est libéré et ajouté aux chunks libres. Chaque chunk a une variable de 64 bits indiquant entre autre son status et sa taille, le status est indiqué par le 64 ème bit de cette variable qui vaux 0 lorque le chunk et libre et 1 lorsqu'il est alloué.
Avant d'expliquer la politique de sélection et donc le recyclage de bloc, il faut savoir que lorsqu'on libère un chunk l'espace mémoire correspondant à ce dernier cherche à fusionner avec les espaces mémoires suivant et précèdent si ils appartiennent à l'allocateur et sont libres. Ensuite seulement il est ajouté aux chunks libres ou rendu au système par munmap.

Fusion des chunks

```
// libération d'un chunk
static void free_chunk(chunk_t *chunk)
{
  if(chunk->previous != NULL && get_status(chunk->previous) == 0){
    del_free_chunk(chunk->previous);

    chunk = merge_w_previous(chunk);
  }

  if(chunk->next != NULL &&  get_status(chunk->next) == 0){
    del_free_chunk(chunk->next);

    chunk = merge_w_next(chunk);
  }

  chunk->size_status = get_size(chunk) | DIRTY_MASK;

  if(only_chunk(chunk)){
    munmap_chunk(chunk);
  }else{
    add_free_chunk(chunk);
  }
}
```

Bien maintenant passons aux chunks libres, il nous fallait une structure de données capable d'accueillir les chunks libres et disponibles dans notre allocateur. Il n'y a que trois opérations à faire sur cette structure de données, ajouter un chunk libre, supprimer un chunk libre, et rechercher un chunk libre d'un taille supérieure à une taille donnée.
Nous avons implémentés plusieurs structure de données mais celle que nous avons retenu est la suivante: nous avons un tableau de 128 listes de chunks, ces listes sont initialement vides, ainsi qu'une fonction prenant en entrée une taille et retournant un entier entre 0 et 127. Cette fonction est telle que lorsque la taille est inférieure à 512 l'entier retourné n'est autre que la partie entière de la taille divisée par 8, sinon l'entier retourné est le log en base 2 de la taille plus 55. Grâce à cette fonction et à cette table il est possible d'ajouter, supprimer et rechercher un élément très rapidement.

Table des chunks libres

```
...
#define f(X)                                ((X < 512) ? (X >> 3) : (log2_fast(X)+55))
...

// ajout d'un chunk libre
static void add_free_chunk(chunk_t *chunk)
{
  size_t index = f(get_size(chunk));

  chunk->next_free = table[index];

  chunk->previous_free = NULL;

  if(table[index] != NULL){
    table[index]->previous_free = chunk;}

  table[index] = chunk;
}

// suppression d'un chunk libre
static void del_free_chunk(chunk_t *chunk)
{
  size_t index = f(get_size(chunk));

  if(table[index] == chunk){
    table[index] = chunk->next_free;
  }
  if(chunk->next_free != NULL){
    chunk->next_free->previous_free = chunk->previous_free;
  }
  if(chunk->previous_free != NULL){
    chunk->previous_free->next_free = chunk->next_free;
  }
}

//recherche un chunk libre suffisament grand pour contenir la quantité de mémoire demandé
static  chunk_t* search_chunk(const size_t size, char clean)
{
  size_t index = f(size);

  while(index < 128){
    chunk_t *chunk_ptr = table[index];
    while(chunk_ptr != NULL){
        if(((clean == 1 && get_dirty(chunk_ptr) == 0) || clean == 0) && get_size(chunk_ptr) >= size){
          return chunk_ptr;
        }
      chunk_ptr = chunk_ptr->next_free;
    }
    index++;
  }
  return NULL;
}
...
```

De cette façon les chunks sont recyclés, après leurs libérations, les durées d'insertion, suppression, et de recherche sont réduites. Pour ce qui est de la politique de sélection, il s'agit du best fit dans la plus part des cas car les chunks d'une taille inférieure à 512 sont triés par ordre croissant. Au dessus de cette taille ce fait est partiellement faux car la liste représentant les chunks d'une même puissance de 2 n'est pas triée par ordre croissants. Globalement la politique de sélection est celle du best fit.

6 Autres optimisations et mutlithreading

Pour ce qui est du multithreading sa gestion est très simple, un mutex est pris et relâché à l'entrée et à la sortie de chacune des fonctions demandées: malloc, realloc, calloc et free.

Gestion du multithreading

```
...
void free(void *ptr)
{
  pthread_mutex_lock(&mutex);

  if(ptr != NULL){
    free_chunk(ptr_to_chunk(ptr));
  }

  pthread_mutex_unlock(&mutex);
}
...
```

Nous avons essayé de faire une deuxième implémentation d'allocateur mémoire inspirée du jemalloc [2] qui est bien plus "thread friendly" car chaque thread a son espace de mémoire, chaque espace mémoire ayant son mutex on évite ainsi la contention sur l'allocation mémoire comme le montre bien la première figure de [2]. Nous avons malheureusement manqué de temps pour finir cette implémentation qui est encore bugger et dont les fonctions calloc et realloc ne sont pas implémentée. Si vous voulez aussi que l'on vous fournisse ce code demander nous.

D'autres optimisations sont effectuées dans l'allocateur. Commençons par calloc, il existe un bit dans la variable servant à contenir la taille et le statuts du chunk qui représente la propreté du chunk, c'est à dire si il à déjà été recyclé ou non. Si un chunk n'a jamais été utilisé alors mmap garantit que toute la mémoire qu'il représente est à 0. Il y a donc trois scénarios possibles lors d'un calloc:
Si il existe un chunk libre et propre compatible avec la taille demandée alors il est retourné.

```
...
if((chunk = search_chunk(total_size,1)) != NULL){
    to_return = chunk_to_ptr(alloc_chunk(chunk, total_size));}
...
```

Sinon si il existe un chunk libre pas propre, compatible et que la taille demandée est inférieure à une certaine limite, alors ce chunk est mis à zéro grâce à memset et retourné.

```
...
else if(total_size < LIMIT_CALLOC && (chunk = search_chunk(total_size,0)) != NULL){
      to_return = memset(chunk_to_ptr(alloc_chunk(chunk, total_size)), 0, total_size);}
...
```

Sinon un nouveau chunk est obtenu avec un mmap, puis découpé si nécessaire, et retourné.

```
...
else{
      chunk_t *chunk = (SIZE_MIN_CHUNK > total_size) ? mmap_chunk(SIZE_MIN_CHUNK) : mmap_chunk(total_siz
```

```
      if(size_small_enough(chunk, total_size))
        add_free_chunk(cut_chunk(chunk, total_size));
      set_status(chunk, STATUS_MASK);
      to_return = chunk_to_ptr(chunk);
    }
...
```

Realloc lui aussi est optimisé, il y 5 scénarios possibles:
Si la nouvelle taille demandée est légèrement inférieure à celle du chunk donné alors rien n'est fait et le chunk donné est retourné.
Si la nouvelle taille est très inférieure à celle du chunk alors le chunk est coupé pour libérer l'espace restant, mis à jour puis retourné.

```
...
if(old_size - size >= REALLOC_MIN)
    add_free_chunk(cut_chunk(old_chunk,size));
...
```

Si l'espace mémoire suivant celui du chunk donné est libre et que la somme de sa taille et de celle du chunk donnée est supérieure à la taille demandée, alors les deux chunks sont fusionnés, le nouveau chunk est coupé si nécessaire puis retourné.

```
...
else if(next_is_free(old_chunk) &&  get_size(old_chunk) + get_size(old_chunk->next) + CHUNK_SIZE >= size
      del_free_chunk(old_chunk->next);

      merge_w_next(old_chunk);

      if(size_small_enough(old_chunk, size))
        add_free_chunk(cut_chunk(old_chunk, size));}
...
```

Si le chunk donné représente la totalité d'un espace mémoire obtenu par un mmap et que mremap est disponible sur le système de l'utilisateur alors le chunk est remapé de façon à ce qu'il fasse la taille demandée et retourné.

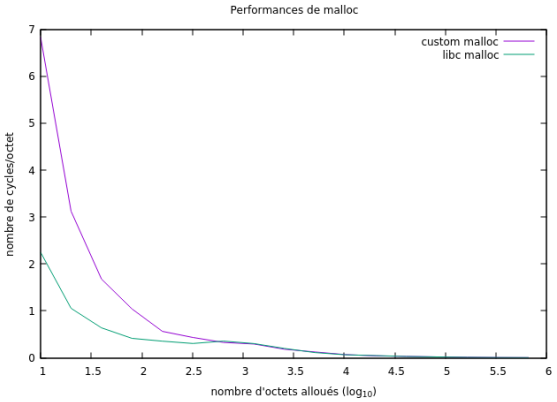
```
...
#if HAVE_MREMAP
else if(only_chunk(old_chunk) && HAVE_MREMAP){
  chunk_t *new_chunk = mremap_chunk(old_chunk, size);

  new_chunk->size_status = size + DIRTY_MASK + STATUS_MASK;

  to_return = chunk_to_ptr(new_chunk);}
#endif
...
```

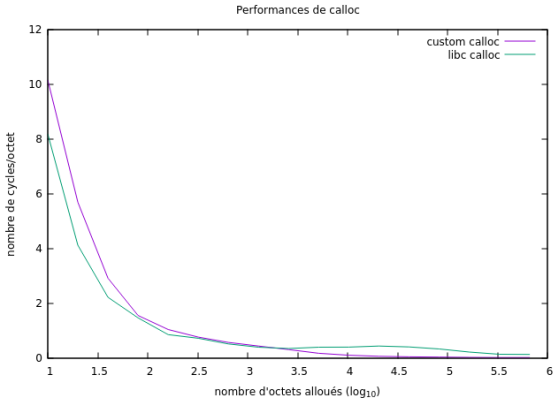
Sinon on effectue un malloc de la taille demandée et on copie le contenu de l'ancien chunk dans le nouveau avant de le libérer.

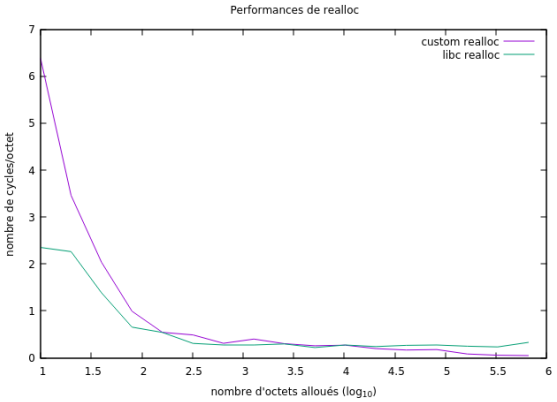
7 Tests de performances et profilage mémoire

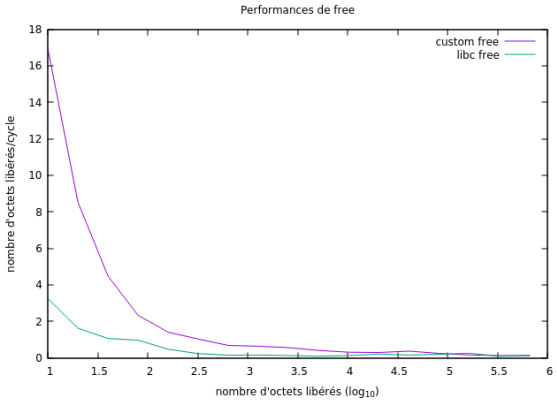
Les compteurs n'apparaissent pas sur le code présenté plus haut pour en faciliter la lecture, mais il est possible dans notre allocateur d'obtenir certaines informations sur l'utilisation de la mémoire grâce à une autre fonction. Sur le dépot ce trouve un dossier perf_test dans lequel ce trouve un fichier "perf_test.c", "test.sh" et "plot.sh". Le programme "perf_test.c" effectue des free, des malloc, des realloc et des calloc d'une taille

choisi aléatoirement et uniformément entre 1 et T un taille donnée à la compilation, puis retourne le nombre de cycles nécessaire pour le traitement d'un octet dans chaque fonctions. Le script "test.sh" fait augmenter cette limite de taille entre 10 et 1000000 octets. Voici les résultats.









On peut voir que pour la fonction malloc nos performances sont très proches de celle de la libc. En revanche pour la fonction free, sur des petites quantités de données à libérer notre free est plus lent que celui de la libc.
Pour les fonctions realloc et calloc si dessus, on observe la même chose que pour la fonction free, c'est à dire que notre allocateur est moins performant pour les petites quantités de données.

```
...
void *print_tacker()
{
  fprintf(stderr, "Ratio de succes sur la recherche: %zu/%zu\n", succes_cpt, fail_cpt);
  fprintf(stderr, "Quantite de mémoire reservé: %zu octets\n", allocated_memory);
  fprintf(stderr, "Nombre de malloc: %zu\n", malloc_cpt);
  fprintf(stderr, "Nombre de free: %zu\n", free_cpt);
  fprintf(stderr, "Nombre de calloc: %zu\n", calloc_cpt);
  fprintf(stderr, "Nombre de realloc: %zu\n", realloc_cpt);
}
...
```

8 Conclusion

Nous arrivons à lancer certains logiciel avec notre allocateur comme openoffice, firefox et la plupart des commandes "classiques" sur nos machines comme ls, apropos, htop, lstopo et autres. Mais néanmoins nous sommes plus lent que la libc, ce qui d'après nos tests de performances viendrait des petites tailles de mémoire. Une de nos hypothèses pour expliquer ceci est que nous n'utilisons pas sbrk qui d'après nos lectures reste utilisé dans la libc pour les petites tailles de mémoire malgré que la fonction soit deprecated. De plus notre gestion du multithreading peut grandement être améliorée notamment en utilisant peut être une approche comme celle du jemalloc. Nous avons cependant beaucoup appris sur la gestion de la mémoire lors de la réalisation de ce projet notamment qu'il n'y a pas une gestion universellement bonne de la mémoire mais qu'il faut s'adapter à chaque besoin pour en tirer les meilleures performances.

References

[1] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[2] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd. 01 2006.

[3] Chuck Lever and David Boreham. Malloc() performance in a multithreaded linux environment. pages 301–311, 01 2000.

[4] Doug Lea. A memory allocation.

[5] Glibc coding team. Malloc implementation in glibc.

[6] Man team. Man page of malloc.

[7] Wikipedia. Wikipedia page of dynamic memory.

[8] Michael Neely R. Wilson, Mark S. Johnstone and David Boles. Dynamic storage allocation: A survey and critical review. *International Workshop on Memory Management*, 1995.